修士学位論文

題目

ソースコードの編集状況に応じた
ソフトウェア部品の自動推薦システム

指導教員
井上 克郎 教授

報告者
島田 隆次

平成 21 年 2 月 9 日

大阪大学 大学院情報科学研究科
コンピュータサイエンス専攻 ソフトウェア工学講座

平成 20 年度 修士学位論文

ソースコードの編集状況に応じたソフトウェア部品の自動推薦システム

島田 隆次

内容梗概

ソフトウェアの構成要素であるソフトウェア部品（クラスやメソッドなど）を再利用することで，ソフトウェアの品質や生産性が向上するといわれている．再利用を促進するために，開発者が明示的に検索を行わずとも自動的にソフトウェア部品を検索する手法が提案されている．この手法では，開発者によるソースコードの編集を監視することで，開発者の手間を増やすことなく，自動的に再利用可能なソフトウェア部品を検索することができる．しかし，その手法では変更を加えずに再利用できるソフトウェア部品のみを検索するため，変更を加えなければ再利用できないソフトウェア部品は発見できない．そのため，再利用の機会が限られるという問題点がある．

そこで本研究では，変更を加えなければ再利用できないソフトウェア部品をも検索できる自動推薦手法を提案する．提案手法ではソースコード中に数多く現れるコメントや識別子に基づき，類似した単語を含む部品を推薦する．その際，曖昧さを許容する検索を行うことで，多少の変更を加えれば再利用できるようなソフトウェア部品も推薦することができる．また提案手法は既存手法と同様に，開発者の手間を増やすことなくソフトウェア部品を推薦することができる．

また本研究では，提案手法を実装したソフトウェア部品自動推薦システム A-SCORE を作成した．さらに，学生を用いた評価実験によって，A-SCORE による再利用の促進などの効果を評価した．評価実験では被験者にプログラミング課題を与え，再利用した既存ソフトウェア部品の個数や，完成したプログラムの不具合の数などを測定した．その結果，A-SCORE を利用したほうが再利用したソフトウェア部品の個数が多く，完成したプログラムの不具合の数も少なくなっており，プログラムの品質は高くなることが分かった．また，既存手法では自動推薦が行えなかったような場合においても自動推薦による再利用が行えた被験者がおり，A-SCORE の効果を示す結果となった．

主な用語

再利用,
ソースコード検索,
統合開発環境への統合,
アンビエントインタフェース

目 次

図 目 次

表 目 次

1　はじめに

ソフトウェア開発においては，適切な再利用を行うことで，ソフトウェアの品質や生産性が向上するといわれている [20]．ソフトウェアの再利用は，ソフトウェア部品と呼ばれるソフトウェアの構成要素であるクラスやメソッドなどの単位で行われることが多い．ソフトウェア部品の再利用を行うことで，既存のソフトウェア部品を新しいソフトウェアの開発に利用し，開発期間を短縮したり不具合を回避したりすることができる．

しかし，ソフトウェア部品の再利用は必ずしも簡単に行えるわけではない．企業で開発したソフトウェアやウェブから入手できるオープンソースソフトウェアなど，ソフトウェア部品は世の中に無数に存在している．しかしソフトウェア部品の中には，ドキュメントも整備されておらず利用方法や動作条件などが分かりづらいために，そのままでは再利用が難しいものが多く存在する．そのようなソフトウェア部品を再利用するためには，開発者がソフトウェア部品のソースコードを読んで再利用できるかどうかを判断しなければならない．しかし大量のソフトウェア部品の中から開発者が手作業で必要な部品を見つけることは手間がかかり困難である．

そこで，大量のソフトウェア部品から目的のものを効率的に見つけるために，キーワード検索によるソフトウェア部品検索システムが主に用いられている．ソフトウェア部品検索システムとは，ソフトウェア部品を機能や用途などで分類・管理して蓄積し，ユーザの指定に応じてソフトウェア部品を探し出すシステムである．多くのソフトウェア部品検索システムでは，キーワード（検索語句）を指定して検索を行うキーワード検索機能を提供している．

しかし，キーワード検索を用いた再利用には 2 つの問題点がある．第 1 に，開発者が意識しないと検索が行われない問題である．第 2 に，開発者が適切なキーワードを選定する必要がある問題である．

これらの問題に対して，Ye らは，開発者が明示的に検索を行わずともシステムが自動的にソフトウェア部品を検索する，ソフトウェア部品自動検索という概念を提案し，その手法を CodeBroker というツールとして実装した [29]．ソフトウェア部品自動検索では，開発者によるソースコードの編集を監視することで，開発者の手間を増やすことなく，自動的に再利用可能なソフトウェア部品を検索することができる．

しかし，CodeBroker には，変更を加えずに再利用を行えるソフトウェア部品しか検索できないという問題がある．ソフトウェア部品の再利用には，変更を加えずに再利用を行う以外にも，再利用先のプログラムに合わせて変更を加えてから再利用する場合や小さなコード片だけをコピー&ペーストにより再利用する場合もある．

そこで本稿では，変更が必要となるようなソフトウェア部品をも自動的に推薦できる，自動推薦の新しい手法を提案する．提案手法では，ソースコード中に数多く現れるコメントや

識別子に基づいて検索を行う．ソースコード中のコメントや識別子を用いることで，コード片の再利用にも対応できるようにした．また，検索の際には，自然言語に対する検索手法である潜在的意味インデキシング LSI(Latent Semantic Indexing)[15] を応用して，曖昧さを許容して検索を行う．こうして，ある程度の差異があるソフトウェア部品も検索できるようすることで，変更を加えてから再利用する場合にも対応できるようにした．

また，提案手法を実装したソフトウェア部品自動推薦システム A-SCORE を作成した．A-SCORE は統合開発環境 Eclipse に組み込んで利用するように作成しており，開発者の活動を邪魔しないアンビエントインタフェースとしての一面を持っている．A-SCORE を利用すれば，普段どおりにプログラム開発を行うだけで,「再利用の機会を逃さない」という自動推薦の恩恵を享受することができる．

最後に，学生を用いた評価実験を行い，A-SCORE による再利用の促進などの効果を評価した．評価実験では被験者にプログラミング課題を与え，再利用した既存ソフトウェア部品の個数や完成したプログラムの不具合の数などを測定した．その結果，A-SCORE を利用したほうが再利用したソフトウェア部品の個数が多く，完成したプログラムの不具合も少なくなっており，プログラムの品質が高くなることが分かった．また，既存手法では自動推薦が行えない場合において，自動推薦による再利用が行えた事例も存在し，A-SCORE の有用性を示すことができた．

以下，2 節では本研究の背景となる既存のソフトウェア部品検索システムについて述べる．3 節では提案手法の詳細について述べ，4 節で提案手法を実装したソフトウェア部品自動推薦システム A-SCORE について説明する．4.3 節では A-SCORE の適用例について説明する．5 節では A-SCORE の評価実験とその妥当性について述べる．6 節では関連研究について述べる．7 節ではまとめと今後の課題について述べる．

2　ソフトウェア部品検索システム

1 節で述べたように，ソフトウェア部品検索システムとは，ソフトウェア部品を機能や用途などで分類・管理して蓄積し，ユーザの指定に応じてソフトウェア部品を探し出すシステムである．現在までに開発されたソフトウェア部品検索システムの例としては，Koders[6] や Google Code Search[4]，SPARS-J[9, 18, 30]，Krugle[7]，Sourcerer[8, 10]，Codase[2] などが存在し，そのほとんどがキーワード検索機能を提供している [17]．ソフトウェア部品検索システムのキーワード検索機能を用いた再利用は，以下のような手順となる．

1. 開発者がソフトウェアを開発しているときに，使えそうなソフトウェア部品を探そうと思い立つ．
2. 開発者が検索のためのキーワードを考え，ソフトウェア部品検索システムに入力する．
3. ソフトウェア部品検索システムがキーワードに合致するソフトウェア部品を検索し，開発者に提示する．
4. 開発者がそのソフトウェア部品が使えるかを判断する．
5. そのソフトウェア部品開発者がそのソフトウェア部品を再利用する．

しかし，キーワード検索を用いた再利用には大きな問題点が 2 つある．1 つ目は，開発者が意識しないと検索が行われないという問題である．上記手順の 1 で開発者が能動的に検索を行わないと，利用可能なソフトウェア部品は検索されない．2 つ目は，開発者がよく知らないソフトウェア部品は検索しにくいという問題である．目的のソフトウェア部品を得るためには上記手順の 2 で適切なキーワードを選定する必要がある．しかし欲しい機能は分かっていてもその機能を表す言葉が分からないなど，それができない場合もある．

これらの問題に対して Ye らは，開発者の指示なしにシステムが自動的に検索を行う，ソフトウェア部品自動検索という概念を提案している [29]．ソフトウェア部品自動検索では，開発者によるソースコードの編集中に，システムが自動的に必要な情報を収集して検索を行い，利用可能なソフトウェア部品を開発者に提示する．ソフトウェア部品自動検索の利点として，次の 2 つが挙げられる．

1. 検索のタイミングをシステムが自動的に決定するため，再利用可能なソフトウェア部品があった場合には開発者が意識していなくてもそれが提示される．そのため再利用の機会を逃さない．
2. 検索条件をシステムが自動的に決定するため，開発者がうまくキーワードを書けなくても検索が行える．

また Ye らはソフトウェア部品自動検索を実現するための手法を提案し，それを実装したシステム CodeBroker を作成している．CodeBroker はメソッドをソフトウェア部品として扱い，メソッドの宣言を書いたときに検索を行う．メソッドの宣言は，メソッドに付随するドキュメントコメント（機能等を説明するコメント）とシグネチャ（引数と戻値の型）を含む．開発者がメソッドの宣言を書くと，CodeBroker はそれに含まれるドキュメントコメントを元に自然言語に対する解析手法である潜在的意味解析法 LSA(Latent Semantic Analysis)[21] を用いて類似メソッドを検索する．その後，検索結果からシグネチャが一致するメソッドだけを抽出し，再利用可能なソフトウェア部品として開発者に提示する．

しかし，CodeBroker には，変更なしで再利用できるソフトウェア部品しか検索できないという問題点がある．これは，CodeBroker がシグネチャの一致による絞込みを行っているためである．シグネチャが一致する，つまり入出力となる引数や戻り値を変更せずにそのまま使えるソフトウェア部品を検索するため，そうではないソフトウェア部品は検索できない．

しかし，ソフトウェア部品の再利用には，再利用先のプログラムに合わせて変更を加えてから再利用する場合や小さなコード片だけをコピー＆ペーストにより再利用する場合もある．大規模なプログラムでは無変更の再利用よりも変更を加えて再利用するほうが多く行われているという報告がある [27]．

また，CodeBroker では，コード片の再利用にも対応できない．これは自動検索を行うタイミングと検索に用いる情報が原因である．CodeBroker ではメソッドの宣言を書いた時点で検索を行うが，コード片の再利用ではメソッドの本体を書いているときに自動検索を行うほうがよい．さらに，CodeBroker ではメソッドのドキュメントコメントとシグネチャだけを用いて検索を行っているが，コード片の再利用にはコード片そのものに関する情報が必要となる．

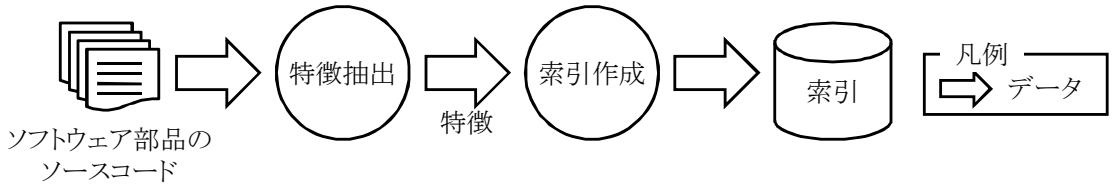

図 1: 索引作成処理の流れ

3　ソフトウェア部品自動推薦手法

本節では提案する自動推薦手法の詳細について述べる．本手法では Java で書かれたクラスをソフトウェア部品として扱う．本手法では，まず，ソフトウェア部品の特徴を元に索引を作成する．ソフトウェア部品の特徴とは，ドキュメントコメントや通常のコメント，識別子を構成する単語である．そして開発者によるソースコードの編集に応じて，自動的に索引を用いて曖昧さを許容した検索を行う．これにより提案手法は変更を加えて再利用する場合や，コード片の再利用にも対応することができる．

本手法では頻繁に検索を行うので，検索の高速化のために索引を用いた検索を行う．そのため本手法は検索に用いる索引を作成する索引作成処理と，開発者の編集に応じて自動的に検索を行う推薦処理からなっている．本手法の処理手順は以下のとおりである．

索引作成処理（図 1）

1. 部品データベースに格納されているソースコードから特徴を抽出する．
2. 得られた特徴から索引を作成する

推薦処理（図 2）

1. 開発者によるソースコードの編集を監視し，検索のタイミングを決定する
2. 編集中のソースコードから特徴を抽出し，検索クエリを生成する
3. 索引を用いて検索クエリに合う類似ソフトウェア部品を検索する
4. 検索されたソフトウェア部品を開発者に推薦する

なお，検索にはベクトル空間モデルの応用である潜在的意味インデキシング LSI(Latent Semantic Indexing)[15] を用いる．LSI を用いることで，表記のゆれや類義語など曖昧さを許容した検索が期待できる．LSI では文章とそれに含まれる単語の出現回数を共起行列として表現するが，本手法では文章にソフトウェア部品を，単語に特徴を対応させている．

以下では各手順について説明する．

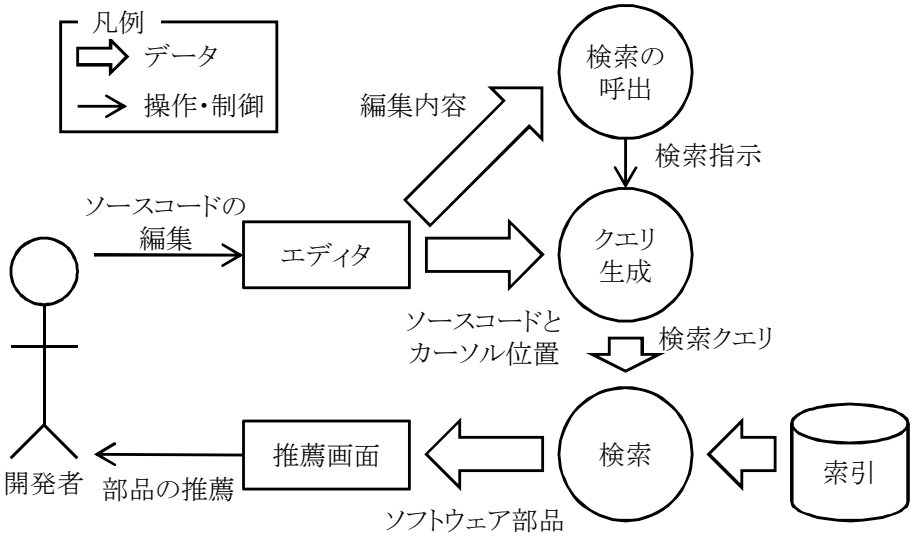

図 2: 推薦処理の流れ

3.1 索引作成 1: 特徴の抽出

検索の実行に先立って索引を作成するために，検索対象であるソフトウェア部品群のソースコードを解析して特徴を抽出する．ここでは，まず特徴について説明し，次に各部品から特徴を抽出する方法について述べる．

本手法で扱うソフトウェア部品の特徴とは，冒頭で述べたようにドキュメントコメントや通常のコメント，識別子を構成する単語である．これらの特徴はソフトウェア部品の機能などを表していると考えられる．識別子はクラス名やメソッド名，フィールド名，ローカル変数名である．ドキュメントコメントや通常のコメントは開発者がそのソフトウェア部品に関する情報を記したもので，機能や用途などコメント外には明示されない重要な情報を持っている．識別子はソフトウェア部品のソースコード全体に偏在しており，その役割や機能に関連した名前をつけることが一般に推奨されている．その中でも特にクラス名，メソッド名，フィールド名はソフトウェア部品の主な構成要素であり，ソフトウェア部品の機能と深い関係があるはずである．また，ローカル変数名は 1 つのメソッドの中という狭い範囲でしか利用されないため，あまり意味のない名前がつけられることもあるが，ソースコード中に最も多く出現する識別子であるため，特徴として利用することにした．

まずソフトウェア部品のソースコードを構文解析し，クラスごとに各要素から語を抽出する．各要素から語を抽出する処理を以下に示す．

通常のコメント まずコメントに含まれる文字列をスペース・ピリオド・カンマ区切りで語に分割する．次に，1 文字だけの語や記号のみからなる語など，意味のない語を除去

表 1: 識別子の分割例

種類	名前	分割結果
クラス名，インターフェイス名	ClassName	Class, Name
メソッド名，フィールド名，ローカル変数名	variableName	variable, Name
定数名	CONSTANT_NAME	CONSTANT, NAME

して特徴とする．

ドキュメントコメント ドキュメントコメントは HTML タグや実体参照 (& など) を含んでいる場合がある．また，本文の他に@記号から始まる@タグを持つ場合がある．そこで，最初に HTML タグや実体参照を除去し，本文のみを抽出して通常のコメントと同様に処理する．

クラス宣言 宣言されたクラスの名前を，CamelCase に従っているとみなして小文字に続く大文字の直前で語に分割する．CamelCase は Java の言語標準で推奨されている命名規則で，複数の単語からなる名前において，各単語の先頭だけを大文字にするというものである．ただし，大文字が連続している部分は頭字語であるので 1 つの単語とする．また，定数名は CamelCase ではなく，全て大文字からなる単語をアンダーバーで繋ぐという命名規則に従うため，それも考慮して分割を行う．分割の例を表 1 に示す．分割された語から，1 文字だけの語などの意味のない語を除去して特徴とする．

メソッド宣言 宣言されたメソッドの名前をクラス名と同様に分割して特徴とする．メソッドが仮引数を持つ場合は，各仮引数名も同様に処理する．

フィールド宣言 宣言されたフィールドの名前をクラス名と同様に分割して特徴とする．

ローカル変数宣言 宣言されたローカル変数の名前をクラス名と同様に分割して特徴とする．

メソッド呼出 呼び出されているメソッドの名前をクラス名と同様に分割して特徴とする．メソッド宣言の場合と違い，メソッドの実引数は処理しない．

次に，抽出した語から検索に有用でない語を除去する．除去する語は以下のいずれかの条件に当てはまるものである．

まれにしか現れない語 1 つのクラスにしか現れない語

```
import java.awt.event.*;

class SelectAllAction implements ActionListener {
  public void actionPerformed(final ActionEvent e) {
    JEditCommanderTable table = JEditCommanderPlugin.leftTable;
    table.selectAll();
  }
}
```

図 3: ソフトウェア部品のソースコードの例 （下線部が抽出される要素）

普遍的すぎる語 半数以上のクラスに現れる語

検索の役に立たない語 情報検索において，一般的に検索の役に立たないと考えられる語を集めたリストが Web 上に公開されている[1]．本手法ではこれに Java の予約語を加えて不要語リストとして利用し，リストに含まれる語を除去する．

最後に全ての語を小文字に変換し，ステミング処理を行って特徴とする．ステミングは語から語幹の部分のみを抽出する処理で，複数形や現在分詞などの語形変化を取り除くために行う．

Java では 1 つのファイルに複数のクラス（ソフトウェア部品）を記述することができるため，各要素がどのクラスに属するのかを決めなければならない．メソッドやフィールドなどはそれらを直接包含するクラスに属することは明白である．ドキュメントコメントは直後の要素を説明すると定義されているため，直後のクラス，または直後のメソッド等が属するクラスに属する．しかし通常のコメントにはそのような明確な基準がない．本手法では通常のコメントはソースコード上でそのコメントを直接包含するクラスに属するとした．また，無名クラスに属する要素は，その無名クラスの外部クラスに属するものとして扱う．

図 3 に示す部品のソースコードから特徴を抽出した例を表 2 に示す．この例では，メソッド名 `selectAll` に含まれる `all` は不要語リストに含まれていたため除去されている．また，メソッドの仮引数名 `e` は 1 文字だけの語なので除去されている．table はステミングによって末尾の e が接尾辞として取り除かれ，tabl になっている．

[1]以下の URL から入手できる不要語リストを用いた．
`ftp://ftp.cs.cornell.edu/pub/smart/english.stop`

表 2: 図 3 から抽出した特徴

特徴	出現頻度
select	1
action	3
perform	1
tabl	1

3.2 索引作成 2: 索引の作成

抽出した特徴を元に，検索に用いる索引を作成する．索引は得られた特徴を LSI の手法に従って変換したもので，ソフトウェア部品名リスト，特徴リスト，索引部品行列，索引特徴行列から成る．LSI ではソフトウェア部品と特徴の関係を，潜在的なトピックを用いてソフトウェア部品―トピックの関係とトピック―特徴の関係に分解する．索引部品行列は，そのうちのソフトウェア部品―トピックの関係を表す行列であり，各ソフトウェア部品が各トピックをどの程度含んでいるかを値として持っている．また，索引特徴行列はトピック―特徴の関係を表す行列であり，各特徴が各トピックにどの程度属しているかを値として持っている．

まず各ソフトウェア部品の特徴を基に，ソフトウェア部品―特徴の共起行列 A を作成する．この行列は列にソフトウェア部品，行に特徴をとる．ソフトウェア部品の数を N，特徴の数を M とすると，A は $M \times N$ 行列となる．行列の ij 要素は j 番目のソフトウェア部品における i 番目の特徴の出現頻度である．また，この行列の列ベクトルを部品ベクトルと呼ぶ．部品ベクトルの次元は M である．

この行列から，LSI の手法に従って索引部品行列と索引特徴行列を作成する．LSI では，まず数学的手法である特異値分解 (Singular Value Decomposition) を用いて，次式に示すように 3 つの行列の積に分解する．

$$A = U\Sigma V^T$$

次に，得られた 3 つの行列から，次式に従って索引部品行列 C と索引特徴行列 F を作成する．

$$C = \Sigma_k V_k^T$$

$$F = U_k$$

但し，Σ_k，V_k^T，U_k はそれぞれ，Σ の最初の k 個の対角要素から構成される $k \times k$ 行列，V^T の最初の k 個の行ベクトルから構成される $k \times N$ 行列，U の最初の k 個の列ベクトルから構成される $N \times k$ 行列である．なお，索引部品行列 C の列ベクトルを索引部品ベクトルと呼ぶ．

3.3 推薦 1: 検索の開始

開発者によるソースコードの編集を監視し，特定のタイミングで検索処理を開始する．

本手法では特徴が変化しない限り検索条件が変化しないため，特徴が変化するタイミングでのみ検索を行いたい．そこで検索を行うタイミングを次のように定めた．

- 文の区切りを表すセミコロンが入力された
- 文の変更後に別の文へカーソルが移動した
- コメントの終了を表す*/が入力された
- コメントの変更後にコメント外へカーソルが移動した

これらをソースコードの編集がある程度完了したタイミングと考え，以下では編集の区切りと呼ぶ．編集の区切りが検出されたら次の検索クエリの生成を行う．

3.4 推薦 2: 検索クエリの生成

編集中のソースコードから特徴を抽出し，検索条件を指定する検索クエリを生成する．検索クエリは組＜特徴，重み＞の集合である．重みはその検索においてその特徴が重要なほど大きな値をとる実数値であり，ソースコード上の現在編集中の位置（カーソル位置）と各特徴の現れる位置が近いほど大きな値をとる．これによって現在編集中の位置に近い特徴を重視した検索を行う．

検索クエリの元になる特徴は，編集中のソースコードを 3.1 節の索引作成 1 と同様に解析することによって得る．ただしこの時，各特徴にはカーソル位置からの距離に応じた重みをつける．なお，同じ特徴が複数回現れた場合，重みはその合計値を使う．

3.5 推薦 3: ソフトウェア部品の検索

3.2 節の索引作成 2 で作成した索引を用いて，LSI の手法に従って検索クエリに合うソフトウェア部品を検索し，順位付けを行う．

まず検索クエリを擬似部品ベクトルに変換する．擬似部品ベクトルは検索クエリに含まれる特徴の重みを，部品ベクトルと同じ順で並べた M 次元ベクトルである．

次に，擬似部品ベクトルに類似した部品ベクトルを持つ部品を，索引を用いて検索する．索引特徴行列と掛けることで擬似部品ベクトルを変換し，変換したベクトルと索引に含まれる各索引部品ベクトルとの類似度を求め，類似度の高い順に数個を検索結果とする．LSI による検索の詳細については付録 A を参照されたい．

表 3: CR+FR 順位付け

部品	FR 値	CR 値	FR	CR	(FR+CR) 値	FR+CR
A	0.997	0.419	1	2	3	1
B	0.952	0.164	3	4	7	3
C	0.982	0.012	2	5	7	3
D	0.874	0.312	5	3	8	5
E	0.901	0.872	4	1	5	2

最後に，LSI の検索結果を元に ComponentRank[18] を用いて順位付けの改善を行う．ComponentRank はソフトウェア部品の重要度を利用関係を元に計算する手法及び計算された値のことである．本手法では特徴を元に LSI によって得られた類似度 (Feature Rank Value, 以下，FR 値) と ComponentRank の値（以下，CR 値）を組み合わせて順位付けを行う．このような複数の基準による順位付けは，複数の検索システムに問い合わせを行ってその結果を統合する，メタ検索システムでよく利用され，いくつかの手法が提案されている．本手法では SPARS-J[30] と同様に Borda の手法 [14] を採用している．Borda の手法では，複数の基準について，それぞれの基準で順位付けを行った結果に対して評価点を割り当て，その合計点を昇順で順位付けすることで最終的な順位を得る．本手法ではこれを次のようにして計算している（表 3）．まず検索結果に対して FR 値による順位付けを行い，その順位を FR とする．同様に CR 値による順位付けを行い，その順位を CR とする．FR+CR の値を計算し，それを (FR+CR) 値とする．最後に (FR+CR) 値を昇順に順位付けすることで最終的な順位 FR+CR を得る．

3.6 推薦 4: 開発者への推薦

検索されたソフトウェア部品の一覧を開発者に提示する．また，開発者が必要に応じてソフトウェア部品の詳細を調べ，再利用を行うかどうかを判断できるようにする．

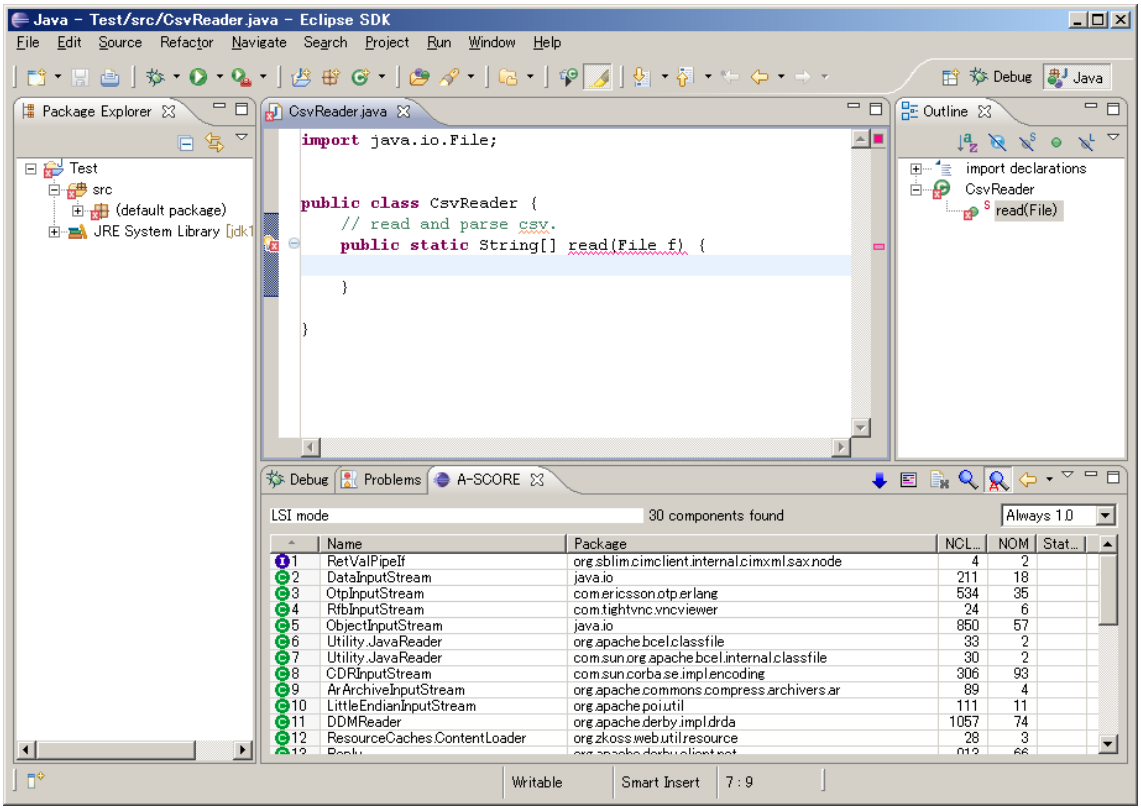

図 4: A-SCORE プラグインを組み込んだ Eclipse の画面

4 ソフトウェア部品自動推薦システム A-SCORE

本研究では，提案手法を実装したソフトウェア部品自動推薦システム A-SCORE (Automatic Software Component Recommendation Environment) を作成した．A-SCORE の動作画面を図 4 に示す．開発者が上中央部のエディタでコーディングを行うと，それに応じて右下部の推薦画面にソフトウェア部品が提示されるようになっている．本節では A-SCORE のシステム構成と実装の詳細，各部の提案手法の流れとの対応について説明する．

4.1 システムの構成

システムの構成と検索処理におけるデータの流れを図 5 に示す．システムはクライアントとサーバから成っており，ソフトウェア部品はサーバ上にある部品データベースで一元的に管理されている．サーバはウェブサービスフレームワークである Apache Axis2[1] を用いてウェブサービスとして実装した．クライアントは統合開発環境 Eclipse[3] を A-SCORE プラグインによって拡張したものである．ソフトウェア部品のソースコードの構文解析には Eclipse JDT(Java Development Tools) の AST(抽象構文木) 生成器を利用している．この

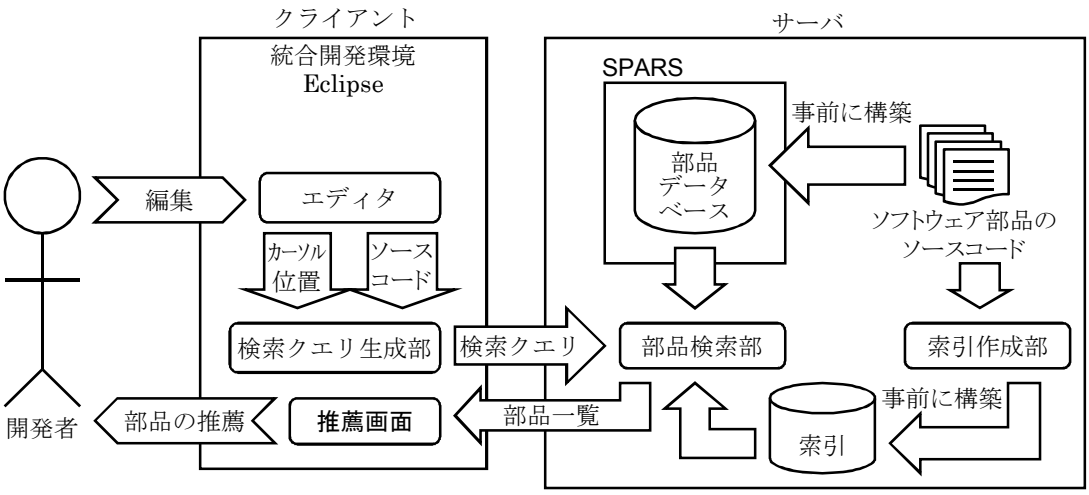

図 5: A-SCORE の構成とデータの流れ

生成器はコメントの情報を含んだ AST を作成することができるため，特徴を抽出するための構文解析器として適している．LSI に必要な行列演算には Java 用の数値演算パッケージである JLAPACK[5] を使用し，特異値分解には SVDPACKC[11, 12] に含まれる las2 アルゴリズムを Java に移植したものを使用した．las2 はランチョス法に基づくアルゴリズムであり，反復回数を減らすことにより結果の精度と引き換えに時間・空間計算量を削減することができるという特徴を持つ．

A-SCORE では開発の効率化や推薦結果の改善のために，内部でソフトウェア部品検索システム SPARS/R を利用している．SPARS/R は SPARS-J の後継として開発されているシステムであり，以下のような機能を持っている．

- ソフトウェア部品の管理
- キーワードによるソフトウェア部品の検索
- ソフトウェア部品の利用関係や被利用関係の解析と表示
- ソフトウェア部品の ComponentRank の計算と表示
- ソフトウェア部品のメトリクス値の計算と表示
- ソフトウェア部品のソースコードのダウンロード

また，SPARS/R は Java で開発されているため，同様に Java で開発した A-SCORE からの利用が容易だった事も SPARS/R を利用した理由の 1 つである．SPARS/R がソフトウェア部品を管理するために持っているデータベースを A-SCORE から参照することで，SPARS/R

の持つ ComponentRank などの機能を A-SCORE から利用することが可能となった．以降，SPARS/R を単に SPARS と呼ぶ．

以下，図 5 の各部について説明する．

部品データベース 蓄積されたソフトウェア部品のソースコード及びそのメタ情報を格納している SPARS のデータベースを合わせて部品データベースと呼ぶ．メタ情報としては以下のようなものがある．

- ソフトウェア部品間の依存関係
  どのソフトウェア部品がどのソフトウェア部品を利用しているのか，などの情報．
- ソフトウェア部品の各種メトリクス値
  ComponentRank や LOC（ソースコードの行数），NOM（メソッド数）など．

索引 部品データベースに含まれるソースコードを解析して作成した，LSI による検索に用いる索引．

索引作成部 手法の索引作成処理 1〜2 を実装している．部品データベースを解析して索引を作成する．

検索クエリ生成部 手法の推薦処理 1〜2 を実装している．開発者によるソースコードの編集を監視して編集の区切りを検出する．編集中のソースコードから特徴を抽出して検索クエリを生成する．検索クエリをサーバの部品検索部に送信する．カーソル位置からの距離による特徴の重み付けには以下に示す複数の重み付け関数を用意しており，利用者が切り替えることができるようになっている．以下の説明で，l_f は特徴の行番号，l_c はカーソル位置の行番号である．

- フラット
  カーソル位置からの距離に関わらず常に 1.0 を用いる．

$$W(f, c) = 1.0$$

- 線形減少
  カーソル位置からの行数に比例して重みが小さくなっていく．最小値が設定されており，一定以上離れると重みは最小値固定になる．

$$W(f, c) = max\{1.0 - abs(l_f - l_c)/\alpha, 0.1\}$$

  $\alpha = 5$ と $\alpha = 20$ の重み付け関数を用意している．

- 二次減少
  カーソル位置からの行数の二乗に比例して重みが小さくなっていく．最小値が設定されており，一定以上離れると重みは最小値固定になる．

$$W(f,c) = max\{1.0 - ((l_f - l_c)/\alpha)^2, 0.1\}$$

  $\alpha = 20$ の重み付け関数のみを用意している．

部品検索部 手法の推薦処理 3 を実装している．索引を用いて検索クエリに合致する部品を検索し，順位付けを行う．検索結果のソフトウェア部品を部品データベースから取得する．そして，ソフトウェア部品一覧をクライアントに送信する．処理の高速化のために，起動時に索引を全てメモリ上に読み込んでおき，動作中には極力ディスクアクセスを行わないようにしている．

推薦画面 手法の推薦処理 4 を実装している．得られたソフトウェア部品一覧を開発者に推薦する．また，ソフトウェア部品の吟味と開発中プロジェクトへの適用を支援するための機能として，いくつかの追加機能を実装している．追加機能については後述する．

4.2 自動推薦以外の機能

ここでは，自動推薦以外の A-SCORE の機能について説明する．

SPARS との連携 推薦されたソフトウェア部品に関する SPARS の詳細情報ページに，1 クリックでアクセスできる機能である．推薦画面に表示されたソフトウェア部品を選択し，詳細表示ボタンを押すと呼び出される（図 6）．この機能を利用すれば，例えば推薦されたソフトウェア部品を利用しているソフトウェア部品を検索することで，サンプルコードを見つけるといった利用方法が可能となる．また，自動推薦で得られた情報を元により適切なソフトウェア部品を明示的に探すために利用することもできる．

ソフトウェア部品のソースコード表示 プロジェクト内のソースコードを表示するのと同じように，ソフトウェア部品のソースコードを手軽に閲覧できる機能である．ソフトウェア部品のソースコードはサーバの部品データベースで管理されており，そのままではクライアント側からは直接アクセスできない．上記の SPARS 連携機能を利用すれば SPARS 上でソースコードを閲覧することはできるが手間がかかる．この機能を利用すれば，推薦画面に推薦されたソフトウェア部品をダブルクリックするだけで，自動的にそのソースコードをサーバから取得し，Eclipse のエディタ上でソースコードを閲覧することができる（図 7）．

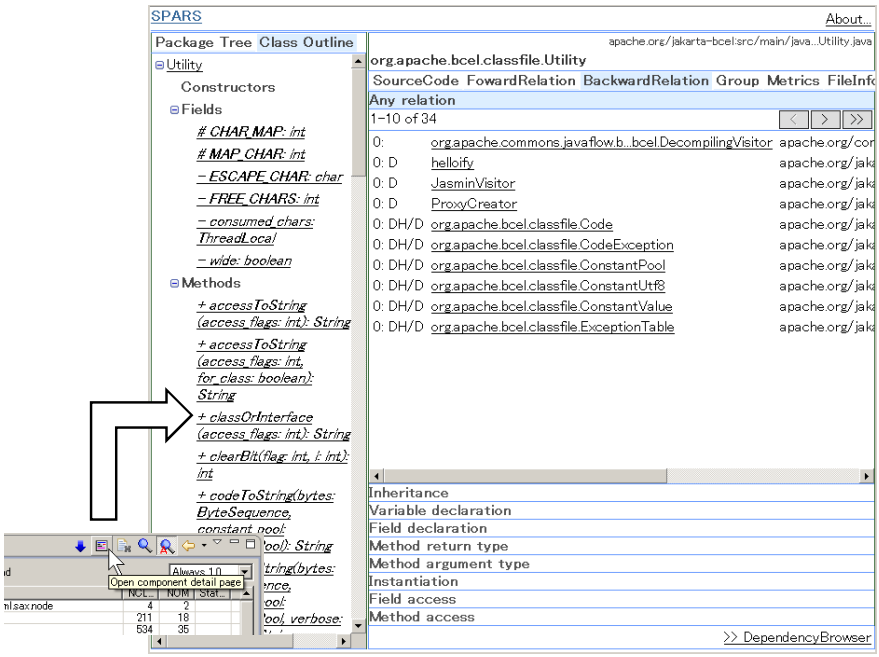

図 6: SPARS 連携機能：詳細表示ボタンを押すと（図左下部），プロジェクトにソフトウェア部品が追加される（図右部）

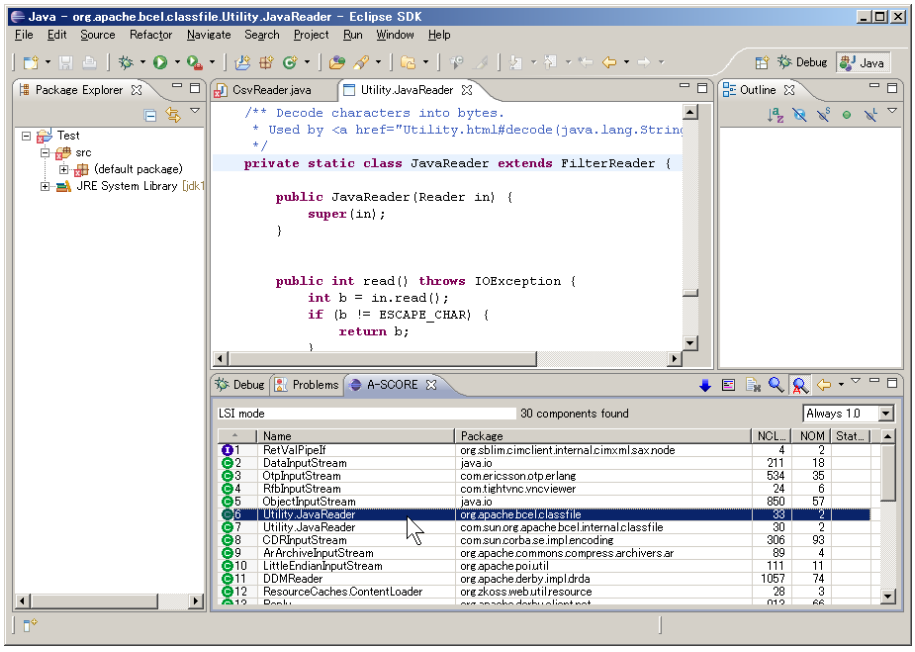

図 7: ソースコード表示機能：推薦されたソフトウェア部品をダブルクリックすると，そのソースコードがエディタで表示される（図上中央部）

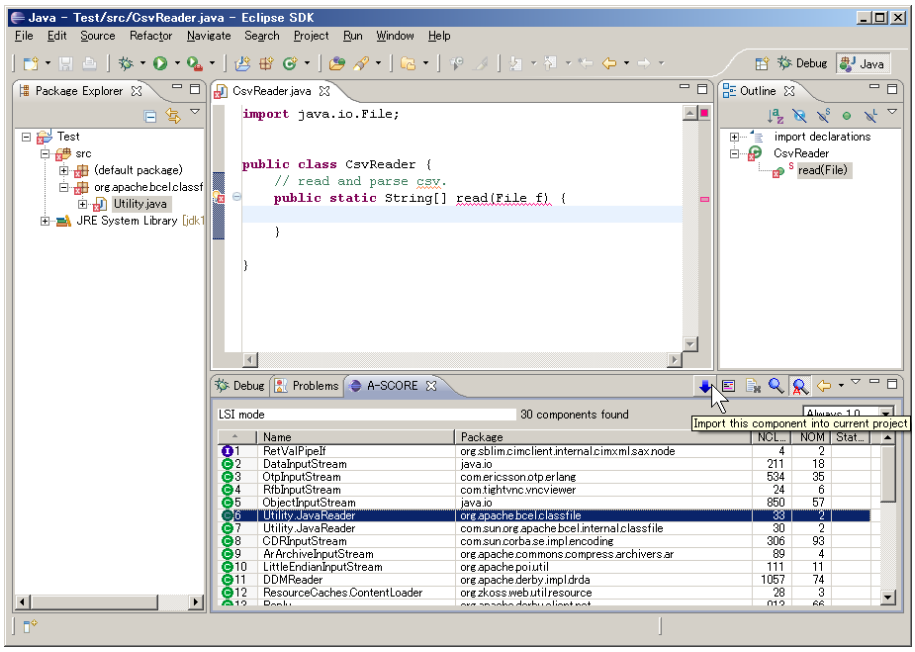

図 8: インポート機能：インポートボタンを押すと（図下部），プロジェクトにソフトウェア部品が追加される（図左上部）

ソフトウェア部品のインポート 1 クリックでソフトウェア部品のダウンロードとプロジェクトへの追加が行える機能である．ソフトウェア部品を部品単位で再利用する場合，再利用に先立ってそのソフトウェア部品を現在のプロジェクトに加える必要がある．前述のとおりソフトウェア部品のソースコードはサーバで管理されているので，それをダウンロードしてプロジェクトに加えるのは手間がかかる．この機能を利用すれば，推薦画面からソフトウェア部品を選んでインポートボタンを押すだけでそのソフトウェア部品を利用することができる（図 8）．

簡易クロスリファレンサ Eclipse によるクロスリファレンス機能と同様の機能を，A-SCORE で表示したソースコード上でも利用できる機能である．クロスリファレンサとは，編集中のソースコード中に存在する識別子からその識別子の宣言箇所にジャンプする機能である．この機能はソースコードの理解にとても有用であるが，Eclipse では，Eclipse が管理しているソースコードに対してしか利用できず，部品データベースに格納されているソースコードや A-SCORE で閲覧中のソースコードに対しては利用できない．この機能は，部品データベース内を検索することで，A-SCORE で表示したソースコード上でもクロスリファレンサを提供することができる．ソースコード上でクラス名を選択し，右クリックメニューから「A-SCORE Search」を選ぶことで，検索が行われ，該当部品が推薦画面に表示される．現在のところこの機能は識別子の中でもクラス名

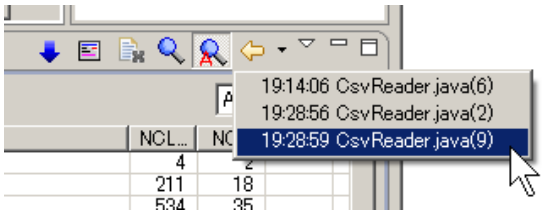

図 9: 検索履歴機能

の参照にのみ対応している．

検索履歴 以前の検索クエリを保存しておき，指示があれば再検索を行うことができる機能である．コーディングを行っていると，直前の推薦結果をもう一度見たいと思うことがある．そのような場合に備えて，A-SCORE では直近数回のクエリを保存しておき，指示された時に再検索を行う機能を提供している．検索履歴には推薦画面右上の矢印アイコンからアクセスできる（図 9）．このメニューには，検索を行った時刻，編集中だったファイル名，カーソルのあった位置が表示される．ここでメニューから項目を選ぶと，その時のクエリが再発行される．

4.3 適用例

ここでは，A-SCORE の適用例を 2 例紹介する．1 つ目は，推薦されたソフトウェア部品が再利用可能であったため，そのソフトウェア部品を修正して再利用する例である．2 つ目は，推薦されたソフトウェア部品そのものは使えなかったものの，そのソフトウェア部品をを手がかりにして，再利用に適したソフトウェア部品を得た例である．

4.3.1 適用例 1：推薦されたソフトウェア部品を直接再利用する例

Java の標準ライブラリに含まれている，ファイルパスを表す `File` クラスのオブジェクトから，そのオブジェクトが表すファイルの拡張子を取得する例を紹介する．`File` クラスにはファイル名を得るメソッドや親ディレクトリ名を得るメソッドは存在するが，しかし拡張子を得るメソッドは存在しない．そのため拡張子を得る必要がある場合には，自分で作成するか既存のソフトウェア部品を再利用しなければならない．

そこで開発者は図 10 に示すソースコードを入力した．このソースコードには `File` 型の変数 `file` と `String` 型の変数 `extension` がある．開発者は `file` から拡張子を取得して `extension` に代入しようと考えている．ソースコードを入力すると，カーソル位置のセミコロンを入力した時点で自動推薦が行われる．開発者は推薦されたソフトウェア部品の名前を見て，8 番目にあった `buildtools.FileUtils` クラスを選んだ．なぜなら，File という名前はファイル処理に関わるクラスであることを表しており，Utils は各所で利用できる便利なメソッドを含むユーティリティクラスで用いられることが多い名前だからである．このソフトウェア部品はオープンソースソフトウェア jEdit のクラスであった．一覧で `buildtools.FileUtils` が表示されている行をダブルクリックすると図 11 のソースコードが表示される．このソフトウェア部品には `String` 型の引数から拡張子を取得するメソッドが含まれており，これが再利用できそうだと判断した．この場合，目的は `File` 型のオブジェクトから拡張子を取得することなので，このソフトウェア部品はそのままでは再利用することはできない．しかしこのメソッドは図 12 に示すように少しの変更で `File` 型の引数を取るようにできるので，そのように変更を加えて再利用を行えばよい．インポートボタンをクリックしてこのソフトウェア部品をプロジェクトにインポートし，変更を加えて再利用を行った（図 12）．図 12 において，再利用に当たって変更した箇所を下線で示している．

なお，CodeBroker はこの場合には自動検索が行えない．CodeBroker はドキュメントコメントを元に検索を行うが，入力したソースコードにはドキュメントコメントの無い `main` メソッドだけしかないためである．また，もし `getExtension` というメソッドを入力して自動検索を行おうとしても，引数が一方は `String` 型，もう一方は `File` 型と異なるため，シグネチャの一致するメソッドのみを提示する CodeBroker では対応できない．

```
import java.io.File;

public class Test {
  public static void main(String[] args) {
    File file = new File(args[0]);
    // get extension of file
    String extension;#
  }
}
```

図 10: 適用例 1 で編集中のソースコード（#はカーソル位置を示す）

4.3.2 適用例 2：推薦されたソフトウェア部品を手がかりにして再利用に適したソフトウェア部品を得る例

ここでは Eclipse の Java 開発環境である JDT(Java Development Tool) の内部でクラスやメソッドなどを表す JavaElement オブジェクトから，その JavaElement が宣言されている位置にジャンプする機能を例に挙げる．これは推薦によって得られた情報を元に適切なソフトウェア部品を得ることができる例である．

開発者は Eclipse の JDT を拡張するためのプラグインを開発している．そのプラグインで，JavaElement オブジェクト（`IJavaElement` インターフェイスで表される）を与えてその JavaElement が宣言されている位置にジャンプする機能を実装する必要が生じた．

まず最初に開発者は Eclipse のキーワード検索機能を用いて再利用可能なソフトウェア部品を検索する．しかし「宣言」を表す”declaration”や”declare”などをキーワードとして検索を行ったが，ジャンプ機能を実現できそうなソフトウェア部品は発見できなかった．

そこで開発者は新しいクラスを作成し，宣言へジャンプする機能の実装を始めた．クラスの作成，メソッドの宣言，コメントの記述などを行うたびに A-SCORE の推薦画面にソフトウェア部品が提示されるので，開発者は時々その画面を見て再利用できそうな部品を探す．この例ではローカル変数宣言を書いたところ（図 13）で再利用できそうなソフトウェア部品 `org.eclipse.jdt.internal.ui.javaeditor.EditorUtility` を発見した．このソフトウェア部品は，そのクラス名より以下のようなクラスであることが分かる．

- パッケージ名の `jdt` より，JDT のクラスであること．
- パッケージ名の `javaeditor` より，Java エディタに関するクラスであること．

```
/**
 *  Misc utils for handing source files.
 */
public class FileUtils {
  ……省略……
  public static String getExtension(String file) {
    int begin = file.lastIndexOf(".");
    if (begin < 0) {
      return null;
    } else {
      return file.substring(begin + 1, file.length());
    }
  }
  ……省略……
}
```

図 11: 図 10 に対して推薦された部品

- クラス名に含まれる Utility より，便利なメソッドを持つユーティリティクラスであること．

開発者はクラス名を見て，このソフトウェア部品が再利用できるのではないかと考えた．

次に開発者は推薦画面に推薦された EditorUtility をダブルクリックしてそのソースコードを表示する．開発者がソースコードを見ていくと，中ほどに「IJavaElement をエディタで開く」旨のドキュメントコメントが付属したメソッド openInEditor を発見した（図 14）．ドキュメントコメントを読んでこのメソッドが利用できそうだと判断した開発者は，EditorUtility を再利用することに決める．この例では現在のプロジェクトが既に JDT を利用しているため，EditorUtility も既にプロジェクトから利用可能な状態になっている．このためインポートする必要はなく，EditorUtility を利用するコードを書くだけで再利用を行うことができる．

しかし EditorUtility は，プラグイン開発者が直接利用すべきではない *internal* パッケージ[2] のクラスである．EditorUtility を利用すれば目的の機能は実現できるが，EditorUtility

[2]Eclipse では，1 つのプラグインで内部的に使うためだけのクラスは，internal という名前のパッケージに含める慣例がある．internal パッケージ内のクラスは他のプラグインから利用すべきではないとされている．

```
/**
 *  Misc utils for handing source files.
 */
public class FileUtils {
  ……省略……
  public static String getExtension(File f) {
    String file = f.toString();
    int begin = file.lastIndexOf(".");
    if (begin < 0) {
      return null;
    } else {
      return file.substring(begin + 1, file.length());
    }
  }
  ……省略……
}
```

図 12: 適用例 1 で最終的に変更して再利用したソフトウェア部品（下線部が変更箇所）

を利用することは望ましくない．そこで開発者は，内部で EditorUtility の機能を利用し，プラグイン開発者が直接利用できるパブリックなインターフェイスを提供しているソフトウェア部品を探そうと試みる．開発者は推薦画面に表示されている EditorUtility の右クリックメニューから Detail を選んで SPARS 連携機能を呼び出し，SPARS の詳細情報ページにアクセスする．そこで EditorUtility を利用している部品の一覧を調べ，その中から *internal* パッケージに含まれていない，直接利用できるソフトウェア部品を探す．この例では org.eclipse.jdt.ui.JavaUI が見つかり，その中のメソッド openInEditor が EditorUtility.openInEditor と同様の機能を提供していることが分かった（図 15）．最終的に開発者はこの JavaUI.openInEditor を利用して目的のジャンプ機能を実現することができた．

```
import org.eclipse.jdt.core.IJavaElement;
import org.eclipse.ui.IEditorPart;

public class JavaElementManager {
  /**
   * Jump to declaration of java element.
   */
  public static void jumpToDeclaration(IJavaElement javaElement) {
    // open file in editor
    IEditorPart editor;#
  }
}
```

図 13: 適用例 2 で編集中のソースコード（#はカーソル位置を示す）

```
public class EditorUtility {
  ……省略……
/**
 * Opens a Java editor for an element such as <code>IJavaElement</code>,
 * <code>IFile</code>, or <code>IStorage</code>.
 * The editor is activated by default.
  ……省略……
 */
public static IEditorPart openInEditor(Object inputElement)
                                          throws PartInitException {
    return openInEditor(inputElement, true);
}
```

図 14: 図 13 に対して推薦された部品の例

```
public class JavaUI {
  ……省略……
public static IEditorPart openInEditor(IJavaElement element,
                boolean activate, boolean reveal)
                throws JavaModelException, PartInitException {
    if (!(element instanceof ISourceReference)) {
        return null;
    }
    IEditorPart part= EditorUtility.openInEditor(element, activate);
    if (reveal && part != null) {
        EditorUtility.revealInEditor(part, element);
    }
    return part;
}
```

図 15: 適用例 2 で最終的に再利用したソフトウェア部品

5　実験

本節では評価実験について説明する．実験の目的は，A-SCORE によるソフトウェア部品の自動推薦によって開発効率，品質，再利用頻度がどの程度向上するのかを調査することである．

5.1　実験内容

実験では，課題を設定し，被験者が課題を完遂するまでの時間を測定した．また，再利用した部品数や完成したソースコードに含まれる不具合の有無を調べた．

課題は，用意した Java のスケルトンコードに，指定した機能を実装することである．なお，A-SCORE ではソースコード中の識別子がクエリとして使われるため，スケルトンコードに含まれるクラス名やメソッド名は意味のない名前にしておき，被験者に名前を付け直させるようにした．この時，被験者は課題の説明資料と Java の API 資料である Javadoc, A-SCORE，SPARS のみを参照してプログラム作成を行うこととした．

なお，比較のために被験者を分けて A-SCORE を用いない場合についても実験を行った．こちらは，被験者は課題の説明資料と Java の API 資料である Javadoc，SPARS のみを参照してプログラム作成を行うこととした．

以下では，課題の内容，被験者，課題の作業環境について述べる．

課題 課題は練習課題 1 種類を含む 3 種類用意した．課題は，スケルトン以外の部分は全て実装済みで，スケルトン部分のみ実装すれば動作する状態のソースコードを用意した．また，スケルトン部分のテストケースを用意し，全てのテストケースに合格することを課題終了の条件とした．

練習課題 画面全体のスクリーンショットを取り，PNG 画像ファイルとして保存するプログラムを作成する．

課題 α 成績情報の記録された CSV ファイルを読み込み，成績上位者の名前を出力するプログラムを作成する．

課題 β ファイルのコピーを行うプログラムを作成する．

なお，全ての課題は適当な入力によって，何らかの形での再利用が可能な部品が推薦されることを確認している．

被験者 被験者は，大阪大学大学院情報科学研究科の学生 3 人と大阪大学基礎工学部の学生 1 人，合計 4 人である．各被験者は学部 3 年次の演習，もしくは各々の研究で Java を扱っており，言語に対する知識は持っている．被験者には実験に先立って事前アンケー

表 4: 実験の被験者と事前アンケート結果（自己申告，5 段階）

被験者	A	B	C	D
Java の熟練度	2	2	3	2
Java のライブラリや再利用に関する経験	1	1	2	1

トを行い，Java プログラミングと再利用に関する熟練度を調査した（表 4）．アンケート結果によって課題の割り当てを調整する予定であったが，今回は各被験者間でほとんど差がなかったため調整は行わなかった．また，事前に A-SCORE の使用方法について講習を行った．被験者を表 5 に示すように分け，実験を行った．

課題の作業環境 課題を行う際に使用する A-SCORE の索引と SPARS のデータベースは以下の内容で作成した．

- JDK1.6 に付属の標準ライブラリのソースコード
- The Apache Software Foundation Projects[3]で公開されているアプリケーション及びライブラリのソースコード
- SourceForge.net[4]で公開されているアプリケーションのソースコード

実験に用いる A-SCORE には，後で作業状況を確認できるように，開発者の作業や検索処理を監視するルーチンを組み込んだ．このルーチンは以下の項目のログを記録する．

- 開発者の編集行為（テキストの挿入・削除・置換）
- 検索クエリ・検索結果
- ソフトウェア部品のソースコードの閲覧
- 画面全体のスクリーンショット（5 秒毎）

5.2 実験手順

1. 最初に全ての被験者が A-SCORE を利用して練習課題を行う．この課題は A-SCORE と SPARS に慣れることを目的としており，作業中にも利用方法や探し方のコツなどについて適宜指導を行った．

[3] http://projects.apache.org/
[4] http://sourceforge.net/

表 5: 課題割り当て

被験者	A	B	C	D
1 番目の課題	課題 α A-SCORE あり	課題 β A-SCORE あり	課題 α A-SCORE なし	課題 β A-SCORE なし
2 番目の課題	課題 β A-SCORE なし	課題 α A-SCORE なし	課題 β A-SCORE あり	課題 α A-SCORE あり

2. 各被験者が表 5 に示された 1 番目の課題を行う．作業中には Java の基本的な事柄（配列の生成方法など）に関する質問のみ受け付ける．

3. 各被験者が同様に 2 番目の課題を行う．

4. A-SCORE で記録したログ，完成したソースコードを調査し，実験結果をまとめる．

5.3 実験結果

実験結果を表 6，表 7，表 8 に示す．以下，各表について説明する．

- 表 6：作業時間
  数値は各項目にかかった作業時間（秒）である．各項目はそれぞれ以下の意味である．

  A-SCORE A-SCORE の操作（主にソフトウェア部品のソースコードの閲覧）を行っていた時間．

  SPARS SPARS でソフトウェア部品の検索を行っていた時間．

  Javadoc Javadoc でソフトウェア部品のドキュメントを閲覧していた時間．

  上記以外 合計時間のうち，上記以外の，コーディングやデバッグなどの作業時間．

  合計 作業全体に要した時間．最初に課題のソースコードを開いた時点から全てのテストケースに合格するまでを測定した．

- 表 7：不具合の有無
  各項目はそれぞれ以下の意味である．

  close 忘れ 入出力処理の最後でファイルを閉じるのを忘れている．

  例外処理忘れ 例外が起きた時にオブジェクトを適切に破棄するための処理を忘れている．

引用符を含む項目に非対応 CSV ファイルでは二重引用符を項目中に含めるためには特別な記述を必要とするが，その処理を行っていない．

バイナリファイルに非対応 ファイルコピーにおいて，ファイルをテキストファイルと仮定して処理を行っており，バイナリファイルを正しく処理できない．

上記の不具合は，Java プログラミングの経験不足や課題に対する理解不足，誤解が原因で起こる．これらの不具合に対するテストケースは「想定していない」「作成が難しい」などの理由で用意していなかった．

- 表 8：再利用部品数
  数値は該当する再利用を行ったソフトウェア部品の数である．1 つの機能を実現するために複数のソフトウェア部品を再利用した場合も，再利用したソフトウェア部品数をそのまま数えている．各項目はそれぞれ以下の意味である．

  インポートして再利用した A-SCORE のインポート機能を用いてソフトウェア部品をプロジェクトにインポートし，そのまま，または改変して再利用を行った．

  コピー＆ペーストして再利用した ソフトウェア部品のソースコードから必要な機能のみをコピー＆ペーストして再利用を行った．

  再利用する部品の手がかりが得られた ソフトウェア部品のソースコードを閲覧することで，再利用すべき部品の存在や名前に気付いた．

  サンプルコードとして再利用した ソフトウェア部品の中に再利用したい別のソフトウェア部品を利用している箇所があり，その部品の利用方法を知ることができた．

5.4 分析と考察

実験結果より，A-SCORE を利用したほうが再利用部品数は多く（表 8），不具合の数は少なくなり（表 7），品質は良くなる傾向にあると言える．一方，作業時間が長くなる傾向にあることが分かる（表 6）．しかし，作業時間に関しても，A-SCORE を用いたほうが結果的に開発が速くなる可能性がある．なぜなら，A-SCORE を利用しなかった場合のプログラムには不具合が多く存在しているため，その後のデバッグや保守に時間がかかることが予想されるからである．

ソフトウェア部品検索システムである SPARS は，A-SCORE を利用するしないに関わらずあまり利用されていない．また，A-SCORE なしの課題では既存のソフトウェア部品を再利用した被験者はいなかった．そこで被験者になぜ再利用しなかったのかをインタビューしたところ，以下のような回答が得られた．

表 6: 実験結果（作業時間（秒））

		A-SCORE あり		A-SCORE なし	
課題 α	被験者	A	D	B	C
	A-SCORE	940	290	—	—
	SPARS	255	0	0	0
	Javadoc	80	505	1225	160
	上記以外	4775	10160	7830	3695
	合計	6050	10955	9055	3855
課題 β	被験者	B	C	A	D
	A-SCORE	940	265	—	—-
	SPARS	75	0	610	0
	Javadoc	790	80	220	1525
	上記以外	4530	1270	1035	4505
	合計	6335	1565	1865	6030
平均		6226.25		5201.25	

表 7: 実験結果（不具合の有無）

		A-SCORE あり		A-SCORE なし	
課題 α	被験者	A	D	B	C
	close 忘れ	あり	あり	あり	あり
	例外処理忘れ	あり	あり	あり	あり
	引用符を含む項目に非対応	なし	あり	あり	あり
課題 β	被験者	B	C	A	D
	close 忘れ	なし	なし	あり	なし
	例外処理忘れ	なし	あり	あり	あり
	バイナリファイルに非対応	なし	あり	あり	あり

表 8: 実験結果（再利用部品数）

		A-SCORE あり		A-SCORE なし	
課題 α	被験者	A	D	B	C
	インポートして再利用した	1	0	–	–
	コピー&ペーストして再利用した	0	0	0	0
	再利用する部品の手がかりが得られた	1	0	0	0
	サンプルコードとして再利用した	0	0	0	0
課題 β	被験者	B	C	A	D
	インポートして再利用した	0	0	–	–
	コピー&ペーストして再利用した	2	0	0	0
	再利用する部品の手がかりが得られた	0	0	0	0
	サンプルコードとして再利用した	1	1	0	0

- 既存のソフトウェア部品があるとは思わなかった.
- 探しても見つからないと思った．それより自分で作ったほうが早いと思った.

この結果から，ソフトウェア部品検索システムが存在してもそれが有効に利用されていないことが分かる．また，A-SCORE ありの課題では再利用が行われていることから，その状況が自動推薦によって改善されていることが分かる.

再利用の方法について見てみると，以下のことが言える.

- インポートして再利用したソフトウェア部品は 1 つしかなく，インポート機能はあまり有効に活用されていなかった.
- 被験者 A が課題 α で行った再利用はソフトウェア部品に修正を加えて行われており，CodeBroker の手法では対応できない再利用の方法であった.
- ライブラリ単位での再利用もサポートする必要がある．被験者 B の課題 β でコピー&ペーストによって再利用された 2 つのソフトウェア部品は，複数の異なる処理を含むユーティリティクラスであった．この 2 つのソフトウェア部品は同じライブラリに含まれていたもので，本来ならライブラリ単位で再利用すべきである．しかし，現在の A-SCORE および SPARS には，ライブラリ単位での再利用を支援する機能は実装されていない．なお，インポートではなくコピー&ペーストで再利用を行った理由は，インポートして再利用するのは依存関係の解決が面倒だったためだと思われる.

5.5 アンケート結果

定量的な評価が行いにくい使い勝手などの項目について定性的な評価を行うために，実験後に簡単なアンケートを行った．アンケートの内容を以下に示す．

1. UI は使いやすかったか
   1（とても使いにくかった）から 5（とても使いやすかった）までの 5 段階評価とした．
2. 推薦結果が役立ったか
   同様に 5 段階評価とした．
3. A-SCORE の各機能は便利だったか
   以下の各機能についてそれぞれ 5 段階評価とした．
   (a) 自動推薦
   (b) メトリクス表示
   (c) ソースコード表示
   (d) SPARS 連携
   (e) 簡易クロスリファレンサ
   (f) インポート
   (g) 検索履歴
4. その他，評価できる点や改善点など
   自由記述とした．

このうち 1〜3 の項目についてアンケート結果をまとめたものを表 9 に示す．また 4 での意見をまとめたものを以下に示す．

- 評価できる点
  - 今まで知らなかった部品が推薦されて便利だった．
  - 自動で部品が出てくるため検索の手間が省けた．
  - 手間をかけずに気軽にソースを見られる点がよかった．
- 改善点
  - コーディング開始時はなかなか使える部品が出てこなかった．欲しい部品が出てくるのが遅かった．

表 9: 事後アンケート結果（5 段階）

被験者		A	B	C	D	最頻値
UI は使いやすかった		4	2	4	4	4
推薦された部品は役に立った		4	4	3	4	4
各機能の評価	自動推薦	4	5	5	3	5
	メトリクス表示	1	1	4	1	1
	ソースコード表示	4	4	4	4	4
	SPARS 連携	2	2	3	4	2
	簡易クロスリファレンサ	3	5	4	4	4
	インポート	5	1	1	4	1
	検索履歴	1	2	1	2	1,2

– ソースコード表示で変数のハイライト機能[5]がほしい.

– 一度閲覧したソフトウェア部品がどれか分かるようにしてほしい.

– ソースコード閲覧時にアウトライン表示がほしい.

自動推薦機能は評価の最頻値が 5 であり，概ね高評価が得られている．しかし，最初のうちは使える部品がなかなか出てこないという意見もある．これは，最初は編集中のソースコードに特徴が少ないため，ソフトウェア部品の機能とはあまり関係のない一部の特徴の影響が大きくなり，LSI による類似部品検索がうまく働かないためだと考えられる.

他にはソースコード表示と簡易クロスリファレンサの機能が高評価であった．これらの機能は編集中のソースコードに対する Eclipse の同様の機能と使い勝手が似ていること，手軽に利用できることが高評価を得られた要因だと思われる.

また，改善すべき点が複数指摘されており，A-SCORE はまだ十分良い使い勝手とは言えない．今後改善していく必要があるであろう.

5.6 内的妥当性

ここでは結果の妥当性に影響を及ぼす可能性のある問題について述べる.

1 つ目は，再利用数増加や品質向上の原因は本当に A-SCORE にあるのかという点である．まず課題の順序の影響が考えられる．つまり後に行った課題のほうが結果がよくなっている可能性である．しかし本実験では順序による影響がでないように，最初に A-SCORE を使

[5]変数を選択すると，同じ変数が使われている場所がハイライトされる機能

用する被験者と最初に A-SCORE を使用しない被験者の両方を用意して実験を行っている（表 4）．さらに，被験者は学部での演習や日々の研究に Java や Eclipse を利用しており，その扱いには既に十分慣れているため，実験の過程で慣れによる影響がでることは考え難い．また，利用するツールを指定したことで，被験者が積極的に A-SCORE を利用し，そのため結果がよくなった可能性についても考える．しかし A-SCORE と同様に利用してもよいとした SPARS はあまり利用されておらず（表 6），A-SCORE の利用時間が長いのはツールを指定したからではなく，A-SCORE の使いやすさがもたらした結果だと考えられる．

2 つ目は，被験者が A-SCORE に慣れていないため操作ミスなどを起こし，A-SCORE を利用した場合の結果が悪くなっている可能性についてである．この点については，被験者全員に対して事前に A-SCORE の使い方について講習を行い，また，他の課題に先立って練習課題を A-SCORE を利用しながら行うことで影響を軽減する措置をとっている．

5.7 外的妥当性

ここでは，実験設定の一般性ついて議論する．

まず，被験者の熟練度に偏りがあるという問題がある．被験者がストリーム処理や例外処理に慣れていないため，再利用を行わずにそれらの処理を自分で書いた場合に不具合が多くなった可能性がある．被験者が再利用を重視した開発に慣れていないため，A-SCORE を用いなかった場合に全く再利用を行わなかった可能性もある．しかし，慣れていない開発者をサポートすることができるという点だけでも A-SCORE は有用であると言える．さらに，熟練者に対しても，当人が熟知していない領域に関するプログラムを書く際には，この実験結果と同様に A-SCORE が役立つと考えられる．なぜなら，熟知していない領域では「簡単に実装できると判断して自作したが，実際には仕様が複雑だったため，既存部品を再利用したほうが速かった」ということが起こり得るからである．

また，複雑なフレームワークを用いた開発の場合の実験が行われていないという問題がある．複雑なフレームワークを用いる場合，再利用なしではプログラムを開発することが困難である．今後，そのような場合に対する評価実験も行う必要がある．

6　関連研究

この節では，本研究と関連性のある既存の研究について述べる．

6.1　ソフトウェア部品自動推薦

Ye らはソフトウェア部品の自動検索手法を提案し，その実装としてソフトウェア部品自動検索システム CodeBroker を開発している [29]．CodeBroker は，Java で書かれたメソッドをソフトウェア部品として，メソッドに付随するドキュメントコメントとシグネチャを元に検索を行う．本手法は Java で書かれたクラスをソフトウェア部品として，クラス内に含まれるドキュメントコメント，通常のコメント，識別子を元に検索を行う．

McCarey らは，アジャイル開発に適したソフトウェア部品自動推薦手法として RASCAL を提案し，Eclipse プラグインとして実装している [23]．アジャイル開発はエクストリーム・プログラミングなどに代表される軽量なソフトウェア開発手法の総称である．アジャイル開発ではドキュメントを書くことを重視しないので，RASCAL はドキュメントを必要としない，利用関係を基にした推薦を行う．RASCAL は協調フィルタリングを用いて推薦を行う．A-SCORE とは異なり，RASCAL では開発者の編集に応じて推薦に用いるデータベースが自動的に更新されるようになっている．

6.2　キーワード検索によるソフトウェア部品検索

ここでは，キーワード検索によるソフトウェア部品検索に関する研究について紹介する．キーワード検索によるソフトウェア部品検索は古くから研究されており，ソフトウェア部品検索システムとして公開されている例が多い．

井上らによるソフトウェア部品検索システム SPARS-J では，キーワード検索による検索結果を改善する手法として，ソフトウェア部品の重要度を表す値 ComponentRank とその計算方法を提案している [18, 30, 9]．本手法による順位付けも，SPARS-J と同様に ComponentRank を用いた改善を行っている．

Bajracharya らは Java クラスをソフトウェア部品としてソースコードの全文検索を行うソフトウェア部品検索システム Sourcerer を提案している [10]．Sourcerer は Google の PageRank[25] を基にした CodeRank と呼ばれる重要度指標と，Fingerprint と呼ばれるソフトウェア部品の特徴ベクトルを用いて検索を行う．

大須賀らは複数のソフトウェア部品検索システムから横断的に検索を行う手法を提案し，プロトタイプとしてソフトウェア部品のメタ検索システム COGEIS を実装している [32]．

その他，キーワード検索によるソフトウェア部品検索システムは Koders[6] や Google Code Search[4]，Krugle[7]，Codase[2] など数多く開発されている [17]．

6.3 キーワード検索によらないソフトウェア部品検索

ここでは，キーワード検索以外の方法を使ってソフトウェア部品を検索する手法を紹介する．これらの手法は，コード片や利用関係，テストケースなどに基づいて検索を行う．

Holmes らは，コード片の再利用を支援する手法を提案し，コード例検索システム Strathcona を開発している [16]．Strathcona は，開発者の要求を表す小さなコード片を入力として，その要求を満たす実行可能なコード例を構造的な類似度を元に検索する．

角田らはソフトウェア部品間の利用関係を元に協調フィルタリングを用いて，入力されたクラスで利用できそうなソフトウェア部品を推薦する手法を提案し，Javawock として実装している [28]．Javawock では，協調フィルタリングによる以下の三つの推薦方法を提供している．

- 入力クラスと同じようなソフトウェア部品集合を利用しているアプリケーションで利用されているソフトウェア部品を推薦する．
- 入力クラスで利用されている各ソフトウェア部品について，同じアプリケーション集合から利用されているソフトウェア部品の集合を求め，その和集合を推薦する．
- 入力クラスと同じようなソフトウェア部品集合を利用しているアプリケーションを推薦する．

また，入力として与えたクラスで既に使われているソフトウェア部品を推薦しないようにすることで推薦結果の質を高めている．

安藤らはソフトウェア部品に要求する仕様をテストケースとして記述したものを入力とし，それに合致するソフトウェア部品を検索するソフトウェア部品検索システム CORST-J を提案している [31]．CORST-J では，ソフトウェア部品の再利用コストをテストケース中に含まれるメソッド呼び出しとソフトウェア部品のインターフェイス間の編集距離とテスト結果の失敗数で定義し，コストの低いソフトウェア部品を検索結果とする．

鷲崎らはソフトウェア部品間の類似度として有向置換性類似度を提案し，入力として与えたソフトウェア部品のプロトタイプから置き換え可能なソフトウェア部品を検索するシステム RetrievalJ を実装している [33]．有向置換性類似度はソフトウェア部品を別のソフトウェア部品と置き換えた時に生じる修正必要箇所の数に基づくもので，類似度が高いほど置き換える際の作業コストが低いことを表している．

Morel らは形式的な仕様記述を元に，要求された仕様を満たすために必要なソフトウェア部品集合を求める SPARTACAS という手法を提案している [24]．

6.4 LSA/LSI を用いたソフトウェア工学手法

川口らは，ソースコード中の識別子をそのソフトウェアの特徴と考え，LSA によって類似ソフトウェアを検索することでソフトウェアのクラスタリングを行っている [19]．本手法では識別子とコメントの両方をそのソフトウェア部品の特徴と考え，類似部品の検索を行っている．

Lukins らは，ソースコード中のコメント・識別子に含まれる単語に基づき，潜在的ディリクレ配分法 LDA(Latent Dirichlet Allocation)[13] を利用してソフトウェアのバグ特定を行う手法を提案している [22]．LDA は LSI を改良した検索手法であり，検索精度などが改善されている．本研究では LSI を利用しているが，LDA への移行も検討している．

7　まとめ

本稿では，ソフトウェア部品を変更せずに再利用する場合だけではなく，変更を加えて再利用する場合やコード片を再利用する場合にも対応したソフトウェア部品の自動推薦手法を提案した．本手法は，ソースコード中に数多く現れるコメントや識別子を利用し，LSI によって曖昧さを許容する検索を行う．また，本手法を実装したソフトウェア部品の自動推薦システム A-SCORE を作成した．

さらに学生を被験者として A-SCORE の評価実験を行った．その結果，A-SCORE を用いることでソフトウェア部品の再利用が促進され，プログラムの品質が向上することが分かった．

今後の課題としては，まず評価実験の充実が挙げられる．外的妥当性の議論で述べた，複雑なフレームワークを利用する場合の実験として，Eclipse プラグイン開発に関する課題も用意することを考えている．また，実験結果の信頼性を向上させるため，人数を増やし，様々な熟練度の被験者を対象に実験を行うことも考えている．さらに，実験の考察でも述べたように，ライブラリ単位での再利用にも対応する必要がある．これにはデータベーススキーマの変更を含む大規模な修正を SPARS に加える必要がある．他には，推薦したソフトウェア部品のソースコードを閲覧する際の利便性の向上が挙げられる．例えば，アンケートで指摘された，ソースコード表示におけるアウトライン機能の実装などを考えている．

謝辞

本研究を行うにあたり，終始，御配慮，御指導，および御鞭撻をいただきました大阪大学大学院情報科学研究科コンピュータサイエンス専攻　井上克郎教授に深く御礼申し上げます.

本研究において， 常に適切な御指導および御助言をいただきました同　松下誠准教授に深く感謝いたします.

本研究において， 常に適切な御指導および御助言をいただきました同　早瀬康裕特任助教に深く感謝いたします.

本研究において， 適切な御指導および御助言をいただきました同　石尾隆助教に厚く感謝いたします.

本研究において， 適切な御指導および御助言をいただきました立命館大学情報理工学部情報システム学科ソフトウェア基礎技術研究室　山本哲男准教授に厚く感謝いたします.

A-SCORE のバックエンドとして利用させていただいているソフトウェア部品検索システム SPARS/R の開発者であり，また本研究において適切な御指導および御助言をいただきました大阪大学大学院情報科学研究科コンピュータサイエンス専攻　市井誠氏に心より感謝いたします.

SPARS/R の前身である SPARS-J の開発に携わった方々に心より感謝いたします.

最後に，その他様々な御助言をいただき，適用実験へ協力していただきました大阪大学大学院情報科学研究科コンピュータサイエンス専攻ソフトウェア工学講座の皆様に感謝いたします.

参考文献

[1] Apache Axis2. http://ws.apache.org/axis2/.

[2] Codase. http://www.codase.com/.

[3] Eclipse. http://www.eclipse.org/.

[4] Google Code Search. http://www.google.com/codesearch.

[5] JLAPACK. http://www.netlib.org/java/f2j/.

[6] Koders. http://www.koders.com/.

[7] Krugle. http://www.krugle.org/.

[8] Sourcerer. http://sourcerer.ics.uci.edu/sourcerer/search/.

[9] SPARS-J. http://demo.spars.info/.

[10] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *Proceedings of Dynamic Languages Symposium*, pp. 681–682, 2006.

[11] M. W. Berry, T. Do, G. W. O'Brien, V. Krishna, and S. Varadhan. SVDPACKC (Version 1.0) User's Guide. Technical Report CS-93-194, University of Tennessee, Knoxville, TN, 1993.

[12] M.W. Berry. Large-scale sparse singular value computations. *International Journal of Supercomputer Applications*, Vol. 6, No. 1, pp. 13–49, 1992.

[13] D.M. Blei, A.Y. Ng, and M.I. Jordan. Latent dirichlet allocation. *The Journal of Machine Learning Research*, Vol. 3, pp. 993–1022, 2003.

[14] J.C. de Borda. Memoire sur les Elections au Scrutin. Histoire de l 'Academie Royale des Sciences, Paris, 1781.

[15] S. Deerwester, S.T. Dumais, G.W. Furnas, T.K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, Vol. 41, No. 6, pp. 391–407, 1990.

[16] R. Holmes. Approximate Structural Context Matching: An Approach to Recommend Relevant Examples. *IEEE Transactions on Software Engineering*, Vol. 32, No. 12, pp. 952–970, 2006.

[17] O. Hummel, W. Janjic, and C. Atkinson. Code Conjurer: Pulling Reusable Software out of Thin Air. *IEEE Software*, Vol. 25, No. 5, pp. 45–52, 2008.

[18] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto. Ranking Significance of Software Components Based on Use Relations. *IEEE Transactions on Software Engineering*, Vol. 31, No. 3, pp. 213–225, 2005.

[19] S. Kawaguchi, P.K. Garg, M. Matsushita, and K. Inoue. MUDABlue: An automatic categorization system for Open Source repositories. *The Journal of Systems & Software*, Vol. 79, No. 7, pp. 939–953, 2006.

[20] C.W. Krueger. Software reuse. *ACM Computing Surveys*, Vol. 24, No. 2, pp. 131–183, 1992.

[21] T.K. Landauer and S.T. Dumais. A solution to Plato 's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge. *Psychological Review*, Vol. 104, No. 2, pp. 211–240, 1997.

[22] S.K. Lukins, N.A. Kraft, and L.H. Etzkorn. Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation. In *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE2008)*, pp. 155–164, 2008.

[23] F. McCarey, M.O. Cinneide, and N. Kushmerick. RASCAL: A Recommender Agent for Software Components in an Agile Environment. In *Proceedings of the 15th Artificial Intelligence and Cognitive Science Conference (AICS2004), Castlebar, Ireland.*

[24] B. Morel and P. Alexander. SPARTACAS: Automating Component Reuse and Adaptation. *IEEE Transactions on Software Engineering*, pp. 587–600, 2004.

[25] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[26] G. Salton and C. Buckley. Term Weighting Approaches in Automatic Text Retrieval. Technical report, Cornell University, 1987.

[27] R.W. Selby. Enabling Reuse-Based Software Development of Large-Scale Systems. *IEEE Transactions on Software Engineering*, Vol. 31, No. 6, pp. 495–510, 2005.

[28] M. Tsunoda, T. Kakimoto, N. Ohsugi, A. Monden, and K. Matsumoto. Javawock: A Java Class Recommender System Based on Collaborative Filtering. In *Proceedings of the 17th lnternational Conference on Software Englneering and Knowledge Englneering (SEKE2005)*, pp. 491–497.

[29] Y. Ye and G. Fischer. Reuse-Conducive Development Environments. *Automated Software Engineering*, Vol. 12, No. 2, pp. 199–235, 2005.

[30] 横森励士, 梅森文彰, 西秀雄, 山本哲男, 松下誠, 楠本真二, 井上克郎. Java ソフトウェア部品検索システム SPARS-J. 電子情報通信学会論文誌 D-I, Vol. 87, pp. 1060–1068, 2004.

[31] 安藤恭平, 金子伸幸, 山本晋一郎, 阿草清滋. テスト実行に基づくコンポーネント検索手法. ソフトウェア工学の基礎 XIII, pp. 119–124, 2006.

[32] 大須賀俊憲, 金子伸幸, 山本晋一郎, 小林隆志, 阿草清滋. ソフトウェア統合検索を利用した再利用支援システム. ソフトウェア工学の基礎 XIV, pp. 203–208, 2007.

[33] 鷲崎弘宜, 深澤良彰. 有向置換性類似度に基づくコンポーネント検索方式の実現と評価. 情報処理学会論文誌, Vol. 43, No. 6, pp. 1638–1652, 2002.

付録

A Latent Semantic Indexing

ここでは，提案手法が検索に利用している LSI と，その前提となるベクトル空間モデルについて説明する．

A.1 ベクトル空間モデル

ベクトル空間モデルでは文書の内容を m 次元空間上のベクトルとしてモデル化する．それによって，ベクトル間の類似度を用いて文書間の類似度を求めることが出来る．

モデル化の手順は以下のとおりである．まず，検索対象となる文書の集合を $D = \{d_1, \ldots, d_n\}$ とする．各文書 d_j を単語の集まりとみなし，d_j に含まれる単語の集合を $W(d_j)$ とする．このとき文書全体の語彙 W は

$$W = \bigcup_{j=1}^{n} W(d_j)$$

と表される．そして c_{ij} を文書 d_j における単語 $w_i \in W$ の出現回数とし，文書 d_j を表すベクトル $\vec{d_j}$ を

$$\vec{d_j} = (c_{1j}, c_{2j}, \ldots, c_{mj})^T$$

と定義する（但し，$m = |W|$）．この列ベクトルを文書ベクトルと呼ぶ．また，全文書について $\vec{d_j}$ を求め，それらを並べると以下のような行列 A が得られる．

$$\begin{aligned} A &= (\vec{d_1}, \vec{d_2}, \ldots, \vec{d_n}) \\ &= \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mn} \end{bmatrix} \end{aligned}$$

この行列 A を文書ー単語共起行列と呼ぶ．

このようにして得られたモデルを用いて，文書間の類似度を求める．ベクトル間の類似度としては内積やコサイン尺度などがあるが，本研究ではコサイン尺度を用いる．コサイン尺度は以下の式で表される．

$$cos(\vec{d_i}, \vec{d_j}) = \frac{\vec{d_i} \cdot \vec{d_j}}{||\vec{d_i}||\ ||\vec{d_j}||}$$

ベクトル空間モデルを用いた類似文書検索は以下のように行う．まず類似文書を検索したい文書 q を d_j と同様に単語の集まりとみなして文書ベクトル $\vec{q}$ を作る．そして各 $\vec{d_j} \in D$ について $cos(\vec{d_j}, \vec{q})$ を計算し，値が 1 に近い文書ほど類似しているとみなす．

A.2 TF-IDF[26]

ベクトル空間モデルを用いた類似度計算では，共起行列 A の各要素として単語の出現頻度をそのまま利用するかわりに出現頻度の正規化を行ったり単語の普遍性を考慮した変換を行うことが多い．単語の出現頻度を単語頻度 TF(Term Frequency) という．単語の普遍性は文書頻度 DF(Document Frequency) で表し，これはある単語が文書全体の中でどれだけ出現したかを反映している．DF が小さい単語は少数の文書にしか現れない，それらの文書に特徴的な単語であるといえる．特徴的な単語を重視すべきなので，DF の逆数である IDF(Inverse DF) を TF に掛けた値 TF $\cdot$ IDF を共起行列の要素の値とする．

本研究では TF には単語の出現頻度をそのまま使い，IDF には以下の式を用いる．

$$IDF_i = \log \frac{n}{n_i}$$

（但し，n_i は単語 w_i を含む文書の数）

A.3 LSI[15]

LSI による検索では上記の手法に加えて，単語間の関連を反映した検索を行うために高次元空間にある文書ベクトルを低次元空間へ射影する操作を行う．

上記の手法で作成した共起行列は要素に 0 が多く，疎なベクトル空間を成している．そのため全く同じ単語を含まない文書は，たとえそれらの文書が意味的に類似していたとしても類似度が 0 になるという問題がある．

LSI ではこの問題を特異値分解 SVD(Singular Value Decomposition) を利用して解決している．SVD は $m \times n$ 行列 A を $A = U\Sigma V^T$ なる 3 つの行列に一意に分解する手法である（但し，$r = min(m, n)$，$U : m \times r$，$\Sigma : r \times r$，$V^T : r \times n$）．これらの行列には “それぞれ上位 k 個の成分のみを残した U_k，Σ_k，V_k^T を掛け合わせて A_k を作ると，A_k は元の行列 A の **rank** k における最小二乗誤差の行列になる” という性質がある．元の行列 A の代わりにこの A_k を用いて類似度計算を行うのが LSI の基本的な考え方である．これによって関連の強い単語はまとめられ，関連の強い単語を含む文書は高い類似度を示すようになる．

しかし，A が疎行列であるのに対して A_k は密行列であり，A_k を求めてから各文書の類似度を求めるのでは膨大な空間計算量が必要になる．そこで LSI では，A_k ではなく，U_k，Σ_k，V_k^T から直接類似度を求める方法が用いられている．コサイン尺度を用いた場合の i 番目の文書 $\vec{d_i}$ とクエリ $\vec{q}$ との類似度の計算は次式で示される．

$$\begin{aligned} cos(\vec{d_i}, \vec{q}) &= cos(\Sigma_k V_k^T \vec{e}_i, U_k^T \vec{q}) \\ &= \frac{(\Sigma_k V_k^T \vec{e}_i)^T (U_k^T \vec{q})}{\|\Sigma_k V_k^T \vec{e}_i\| \|U_k^T \vec{q}\|} \end{aligned}$$

ここで，$\vec{e}_i$ は i 番目の次元に対応する要素のみが 1 である n 次元の単位ベクトルである．

なお，本研究では $k = 500$ としている．